Distributed Implementation of Coordinated, Network-Wide Policies and Protocols with FlowFlex

Paper #163 — 14 Pages

ABSTRACT

The increasing programmability of network devices gives protocol designers and network operators considerably more flexibility in defining custom protocols and traffic processing functions. Today, network operators and protocol designers have the option of either operating at flow-level granularity, which offers coordinated control; or packet-level granularity, which offers flexibility, but not coordinated control. Today's network programming paradigms force operators to choose between the fine-grained control and expressiveness of packet processing and the coordination of flow processing, which makes it difficult to quickly realize a distributed implementation of a global, network-wide policy. Designers must also choose between the flexibility of hardware-based solutions and the fast development cycles offered by software. This paper proposes a system called *FlowFlex* that offers network designers the best of both worlds: with FlowFlex, operators can quickly design, implement, and deploy network systems and protocols that offer fast, distributed, implementations that require coordinated control *and* fine-grained operations on packets. We present the design and implementation of the FlowFlexframework and show how it can improve both expressiveness and efficiency for three real-world networking applications.

1. Introduction

Recent years have seen an explosion of proposals for custom processing and routing of traffic in communications networks, both in the wide area and within enterprises and data centers. Ideally, designers of these technologies would like to be able to design, implement, and deploy protocols that can both realize coordinated, network-wide policy and operate on fine-grained traffic information at hardware forwarding rates. Unfortunately, these goals have proved to be at odds with one another: designers must typically choose between coordinated control over traffic (which is suitable for specifying configurations and policies) and fast, fine-grained control (which is necessary for deployment in practice). This paper presents a model for programming networks, FlowFlex, that attempts to better balance these concerns.

Software-defined networking—the ability to specify various traffic processing functions in software or programmable hardware, rather than in custom ASICs—gives developers, network operators, and protocol designers the opportunity to customize network protocols. This approach allow to customize how network devices process traffic as it traverses the network. This approach is evident in various burgeoning networking technologies, including the emerging OpenFlow standard [2], the Click modular router [14], network processors, and programmable network FPGAs [17]. These technologies allow both operators and researchers to design, prototype, and evaluate "clean slate" networking protocols and techniques before they are ultimately incorporated into custom hardware and deployed across the network.

Although software-defined networking allows developers to "program" specific functions into network devices, this capability begs the question of what programming paradigms and platforms strike the right balance between the ability to specify and implement coordinated network-wide control, the ability to perform fine-grained operations, and the ability to quickly deploy the resulting solutions on network devices that can process packets at line rate. Existing programming paradigms typically force programmers to choose between fine-grained, packet-level control that is often implemented directly on network devices (e.g., in ASICs) or as specialized programming platforms (e.g., Click [15], NetFPGA [17]), which make it difficult to implement a global, network-wide policy; or via centralized, coarse-grained control platforms (e.g., 4D [12], RCP [4], OpenFlow [2], NOX [1]), whose architectures make it inherently difficult to perform fine-grained, packet-level operations at line rate. As a result of this difference, developers are forced to choose between the convenience and fast development cycles offered by software and the sheer forwarding performance that hardware can provide.

To better appreciate the shortcomings introduced by current programming paradigms and platforms, consider a problem that our campus network operators face today: these operators would like to rate-limit voice over IP (VoIP) calls for specific *users* (not devices) on the network. Doing so involves: (1) inspecting the packet payloads during the initial application-level handshake to associate a flow with a particular user; (2) installing flow-based rules—coordinated across the forwarding devices in the network—that rate-limit the traffic for the flows corresponding to that user. These example applications suggest instead a *hybrid* programming paradigm, whereby network policies and traffic processing can be expressed and

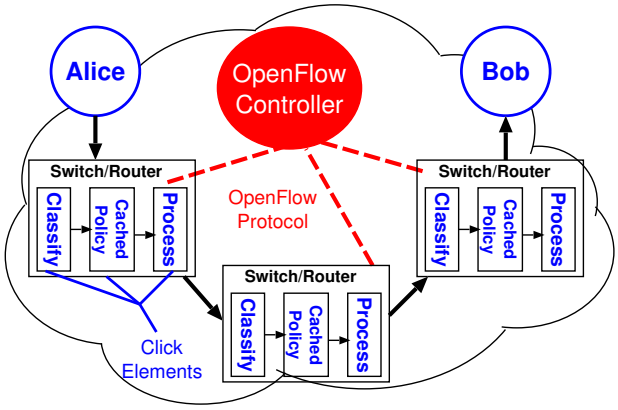

Figure 1: FlowFlex allows developers to express global per-flow policies and routing decisions (using OpenFlow) but enforcement is handled locally per-packet (using Click). In addition to reducing development effort, this hybrid architecture permits solutions not practical in the pure per-flow or per-packet worlds.

implemented with a combination of distributed packet-level processing (to perform classification and attribution of flows to specific users) and flow processing (to implement the per-flow rate-limiting). None of the existing programming paradigms offer the combination of fine-grained control, coordinated network-wide policy, forwarding performance, and fast development time that this application requires.

This paper presents a programming paradigm, *FlowFlex*, that allows protocol designers and developers the ability to design and implement protocols that require *distributed, fine-grained control of global, coordinated policy*. FlowFlex combines the advantages of both packet and flow processing in specifying and implementing network protocols and other mechanisms for manipulating traffic (Figure 1). The programming model also allows the designer to specify global policy, but implement aspects of these policies distributed across the network devices. In essence, FlowFlex offers the best of both words: operators can get the speed and flexibility offered by implementing policies on distributed network devices with the coordination offered by a central control model.

The FlowFlex programming model centers around a pipeline that decomposes forwarding into three distinct operations, as shown in Figure 2: Classification, policy matching, and processing. *Classification* constitutes a set of rules for assigning subsets of traffic into different classes; these rules are defined and managed centrally, and may constitute a combination of packet and flow-based rules (*e.g.*, identifying a user in a VoIP session). *Policy matching* determines what processing primitives should be applied to traffic that matches a specific class of traffic; a central controller associates each class of traffic with a policy and exports the appropriate policy to the switch

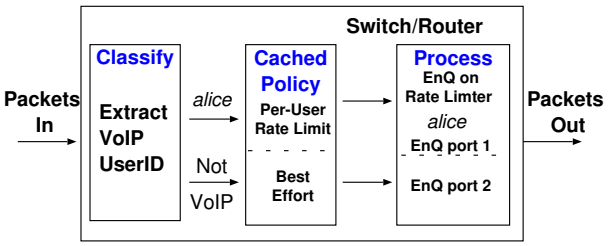

Figure 2: FlowFlex decomposes packet forwarding into three tasks: classification, policy matching, and processing.

as a logic block that comprises one or more flow-table rules (*e.g.*, apply per-user rate limits). *Processing* applies the logic block to the appropriate set of traffic; this block may take specific actions, such as dropping or rate-limiting or forwarding on a specific port. Processing may also be per-packet or per-flow. We use OpenFlow to implement flow processing and classification, Click to implement packet processing and classification, and a new Click element that performs flow based processing (the "`OpenFlowClick`" element) [21] as the glue between these two programming models.

FlowFlex's programming paradigm presents many possibilities for new classes of traffic-processing applications. To explore and evaluate the utility of this new programming model, we use FlowFlex to design, prototype, and evaluate three example applications:

- distributed malware classification and suppression,
- multipath routing with duplicate elimination, and
- user-specific quality of service.

For each of these case studies, we explore how current networking technology solves the problem and how FlowFlex can improve the state of affairs.

The rest of the paper is organized as follows. Section 2 discusses existing programming paradigms in more detail, and explains how these paradigms fail to satisfy various desiderata for a programming model. Sections 3. 4, and 5 provide an overview of the FlowFlex design, architecture, and implementation, respectively. Sections 6–8 show how FlowFlex can be used to easily implement and deploy three network technologies that are difficult to deploy using existing technology. Section 9 discusses various extensions to FlowFlex, and Section 10 concludes.

2. Network Programming Paradigms

The increasing programmability of network devices and systems offers great hope for the rapid deployment of new networking technologies, but these new opportunities also beg the question of what programming model is appropriate for any particular task. To better understand this challenge, we characterize the design tradeoffs of various programming models, offering a survey of related work.

Our goal in the design of FlowFlex is to provide a programming model that is flexible enough to solve a wide array of practical problems while at the same time sparing developers from debugging and deployment hassles.

We first describe various design tradeoffs, contrast existing network programming paradigms, and characterize how they reflect points in this tradeoff between flexibility and complexity. There are a variety of possible programming models for realizing any of these applications, but existing approaches cannot achieve fine-grained coordinated control, fast forwarding performance, *and* fast development time.

> **"Fine-Grained Control, Coordinated Control, Fast Forwarding, Fast Development: Pick Two"**

Table 1 summarizes these tradeoffs, which we will explore them in detail in this section. A network designer's challenge is to find a programming paradigm that offers fine-grained, coordinated control over forwarding, is fast enough to forward packets at line rate, and yet is simple enough that the debugging and innovation cycle is not prohibitively long or costly. Consider the task of designing, deploying, and evaluating a new network protocol or application. For example, later sections of this paper expound on three example applications in detail: per-user, application-level traffic classification; distributed intrusion detection based on packet-level signatures; and network-level redundancy elimination. All three of these applications require some amount of a *fast, packet-level distributed implementation of a global policy*. In the remainder of this section, we will explain how existing programming paradigms fall short of achieving all three of these design goals.

Centralized controllers: Coordinated Control & Fast Development Network programming models can be either centralized or distributed. Typically, a designer wants to develop a specific set of primitives that enforce some global property across the network as a whole. A tension exists between the ability to enforce some global, coordinated policy and the ability to implement that policy in a distributed, fine-grained fashion on the network devices themselves. The designer of a routing protocol (or the network operator) may want to create a forwarding rule that produces a desired global correctness property: for example, the operator may want to enforce that forwarding takes the shortest path, is loop-free, and resilient to link failures. Enforcing these global properties by examining the local device configurations, however, has proved to be quite difficult in practice [9, 18].

Given the complexities involved in creating and correctly implementing distributed protocols [6, 22], centralized logic is naturally more desirable and has given rise to various architectures that control the network from a logically centralized entity [12, 13]. These protocols can offer network-wide control and fast development, but they typically do not offer fast forwarding performance: for example, centralized architectures such as Ethane [5] require much of the network traffic to be forwarded through a centralized controller, which creates a performance bottleneck.

Another instantiation of this approach is the OpenFlow/NOX [1] model. OpenFlow is a flow-based API to control how packets are forwarded [13, 20]. Forwarding decisions are made by a centralized OpenFlow *controller* and then cached on the switch or router in the form of *flow entries*. OpenFlow makes decisions per-flow as opposed to per-packet. The centralized logic of the OpenFlow controller combined with the cached rules makes developing new protocols with OpenFlow fast without compromising efficiency. Unfortunately, OpenFlow has limited classification capabilities, so it does not satisfy the goal of fine-grained control. OpenFlow polices can only take action based on 10 pre-defined packet header fields. It is possible to inspect packet payloads in OpenFlow by routing packets through the centralized controller, but this approach may have poor scalability and performance. Openflow offers only a limited set of packet processing actions.

Software routers: Fast Development & Fine-Grained Control The Click software router is another paradigm for deploying new network protocols [15]. Because Click is fully programmable at the packet level, it allows the protocol designer to define arbitrary classification capabilities, policies, and packet processing. However, Click's logic is implemented on individual packet forwarding devices (e.g., routers), and the logic is limited to operating on a packet-processing control flow. Its inherent decentralization and focus on packet level operations may make enforcing global properties difficult in practice. Additionally developing and deploying new network protocols and applications is fast with software routers such as Click, but the performance is inherently limited by software. Thus, software routers can offer fast development and fine-grained, packet-level control, but they do not provide fast packet forwarding or the opportunity for coordinated control.

Custom hardware: Fine-grained Control & Fast Forwarding Application-specific integrated circuits (ASICs) offer arbitrary control but they are costly, both in terms of money (switch and routers cost between thousands to millions of dollars each) and time (ASIC development is typically a multi-year process). Additionally, as multiple revisions of the ASIC design may be required, the development cycle is likely to be long. Worse yet, the resulting deployment will not be flexible—as new technologies develop, or as network operators change their policies or the

	Coordinated Control	Fine-Grained Control	Fast Forwarding	Fast Development
Centralized control (e.g., 4D)	✓	×	×	✓
Software routers (e.g., Click)	×	✓	×	✓
Custom hardware (e.g., ASICs)	×	✓	✓	×
Prog. h/w (e.g., NetFPGA)	×	✓	✓	×
Network configuration	×	×	✓	✓
FlowFlex	✓	✓	Possibly	✓

Table 1: Existing programming paradigms cannot achieve fine-grained network-wide control, fast packet forwarding rates, and fast development cycles. The goal of FlowFlex is to offer all three.

types of policies that they wish to implement, the ASICs will remain fixed.

Programmable Hardware: Fine-Grained Control & Fast Performance Network processors [25] and NetFPGAs [17] provide options to process traffic with much finer granularity. Development of anything moderately complex on NetFPGAs, however, requires significant development and debugging time. The limited circuit area also constrains the amount of functionality that can reside on the board itself. Network processors have specialized instructions to operate on the network data, but programming such chips requires that the researchers know about low level details of the instruction set and programming the chip itself, so development cycles can be long. In addition, developing portable applications to work on different vendor platforms is challenging. As with Click, programmable network hardware is designed to operate on individual network devices, so this programming model has no notion of coordinated, network-wide control.

Network configuration: Fast Development & Fast Forwarding Routing equipment vendors expose a limited set of controls to affect traffic forwarding [7]; thus, another paradigm is controlling the network by adjusting network configurations alone. This paradigm, of course, offers immediate deployability but almost no control—the example applications we mentioned above cannot be implemented using vendor configurations alone. An alternate approach would be to implement the protocols entirely in software, which would afford a significant amount of control and flexibility and short development time, but would not achieve the speed afforded by hardware.

3. FlowFlex Design

We outline the design of FlowFlex and highlight three new features of FlowFlex's programming model.

3.1 Requirements and Overview

Overview FlowFlex allows protocol designers to leverage both OpenFlow's high-level rapid prototyping flow-processing and Click's low-level detailed packet-processing. FlowFlex consists of a centralized controller that receives updates from switches across the network. The controller maintains a consistent view of the entire topology and network policies. Programmable switches act as distributed points for monitoring traffic and for taking specific processing actions on the traffic. Each switch has a table with rules and actions for traffic that matches those rules; these actions might include forwarding the packet, modifying headers, or executing customized code on the packet. These customized modules can modify both packet headers and contents. The rules and actions along with customized code modules are loaded from a centralized controller. Because the centralized controller keeps a consistent, globally coordinated view of the network and implements a network-wide policy from a central location, it can make decisions about how to treat each traffic flow more effectively and consistently.

FlowFlex has the following design characteristics:

- *Hybrid packet and flow processing.* FlowFlex can implement policies on either a packet or a flow-level granularity, or on some combination of the two. Specifically, the switches can perform both classification and processing at both the packet and flow granularity.
- *Programmability.* The switches that forward packets classify and process traffic at both the packet and the flow level. These switches can classify and process traffic based on logic that is flexible enough to express a variety of policies, yet constrained enough to avoid introducing security problems.
- *Optimization.* FlowFlex allows switches to cache the results of traffic forwarding decisions for flows or groups of flows.

3.2 Design Features

We now describe the new features that FlowFlex offers. FlowFlex provides flexibility and programmability at each of three stages: classification, exception handling, and processing. We briefly describe these features below.

Enhanced classification When performed at switches themselves, traffic classification is typically limited to features that are a function of flow statistics or fields in packet headers. FlowFlex permits traffic classification

that is based on more fine-grained or detailed features—for example, some coarse-grained classification might be done based on contents of packet payloads, counts of traffic volumes, and so forth. Although some aspects of traffic classification could be performed across the distributed network devices, deploying complex deep packet inspection logic at each of these switches would be costly. In contrast, FlowFlex permits coarse-grained classification based on packet-level characteristics, leaving more precise implementation of policies to subsequent steps in the pipeline. In Section 6, for example, we explain how FlowFlex can provide enhanced quality of service features, such as mapping traffic flows to different rate-limiting policies depending upon the user associated with the traffic flow.

Policy exceptions Although many policies can be expressed as flow-table rules, some policy, may require more detailed examination of portions of traffic traces. As an example, a network operator may wish to implement a certain policy for a class of flows (e.g., all flows that match a certain string towards the beginning of the flow). FlowFlex can implement these policy exceptions by forwarding all traffic that does not match an existing flow table rule to a central controller. That controller can then take the appropriate action and install the corresponding logic block for that flow at the switch, either as a flow-table rule or as additional custom processing elements (which we describe next). In Section 7, we explain how FlowFlex can implement an efficient, yet coordinated, distributed intrusion detection system using policy exceptions.

Custom processing FlowFlex also allows a switch to store logic elements that perform custom processing of flows. Current flow-processing models such as OpenFlow only permit a limited set of actions (i.e., drop, forward, and send to controller), but many policies may require more sophisticated actions. For example, a richer, more expressive set of policies might include the ability to tunnel a packet to an arbitrary location or encode a packet stream with error correction. In Section 8, we explain how FlowFlex's custom processing capability can be used to implement a multipath routing scheme with automatic duplicate packet generation and suppression.

Returning to Table 1, we see that FlowFlex can achieve many of the desired goals. The ability of FlowFlex switches to redirect traffic to a centralized controller allows an operator to implement centrally coordinated policy. The classification and application of custom processing on the switches themselves allow for fine-grained control. The ability to push centrally coordinated decisions back to FlowFlex through logic blocks in some cases enables the implementation of these policies in fast hardware (e.g., if the logic block can be encoded in flow-table rules). Finally, the ability to express all of these functions as a combination of Click configuration and flow-table entries makes development fast.

4. FlowFlex Architecture

This section describes the FlowFlex architecture. We describe the three stages of the FlowFlex switch pipeline—classification, policy matching, and processing—and explain how this pipeline allows a range of policies to be realized. We then illustrate these steps in the context of an example scenario.

4.1 Overview

The FlowFlex architecture allows network operators to realize flexible policies for small to medium-sized networks, such as enterprise networks. The architecture is based on the following observation: by and large, network operators may wish to apply policies across the entire network, but these policies may depend on packets, which are best processed at the switches themselves (i.e., either packet-level statistics, or contents of packet payloads).

Figure 4 shows the FlowFlex architecture. The system has two main components: (1) FlowFlex-enabled switches, and (2) a logically centralized controller. The switches perform packet classification and processing but rely on a centralized controller to perform network-wide policies. This split allows the FlowFlex architecture to achieve *distributed implementation of centralized policy*.

4.1.1 *Controller function*

Most of the traffic processing and forwarding is performed at the switches; this processing may be performed at either the packet level or the flow level. The controller deploys rules corresponding to network-wide policy in switches that are distributed across the topology; it converts the policies into rules that can be installed in switches. Based on the switch's initial classification of a traffic flow, the controller determines which policy should apply to that flow, translates that high-level policy into the appropriate logic blocks that can be installed on the switches, dispatches these logic blocks corresponding to that policy to one or more switches across the network, and relies on switches to execute the rules corresponding to that policy.

4.1.2 *Switch pipeline*

Recall that each FlowFlex-enabled switch implements the following functions, as shown in Figure 2:

Classification: The central controller installs some set of rules in the switch's internal classification table. Based on these rules, the switch classifies the traffic according to the appropriate traffic category by matching the packets headers, statistics about the flow, or possibly data with predefined rules. The switch first tries to classify the traf-

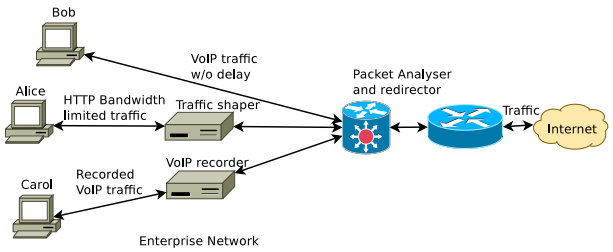

Figure 3: Comparison with current state-of-the-art solutions. Each separate functionality (DPI, shaping, recording) requires a specialized ASIC based solution

fic using rules that apply only to the packet headers. If that fails, then the rules that require packet content inspection are applied. If packet is still not classified then it is sent to the controller to detect its class and corresponding logic block. The controller will match the packet to the policies and send back that information to the switch. Thus, further traffic can avoid deep packet inspection.

Cached policy & logic block table: This table stores the classification rules and logic blocks to be applied to the packets.

Processing: After classifying the packet, the switch processes the packets according to the corresponding action. This processing might vary in complexity, ranging from simply forwarding the traffic to an output port (which could, for example, be implemented as a simple OpenFlow rule) or as complex as encrypting the packet contents before forwarding (which would require more complex packet processing at the switch). The controller installs logic blocks corresponding to these primitives and can load and unload them dynamically. The controller can also define specialized *exception conditions*. Whenever these exception conditions are satisfied, the packet is sent to the controller.

Some examples of an exception could be "Every 50th packet of a TCP flow" or "Any packet containing regular expression regex". Exceptions allow execution control to be transferred to the controller after facing some event. In that case, the controller might decide to replace the current execution block for that flow with a new one. After processing the packets, the switch forwards them through appropriate ports or feeds the packet back to the classification phase. With the feedback loop, it is possible to process packets with different logic blocks in each iteration of pipeline traversal.

4.2 Example Scenario

In this section, we show how the FlowFlex architecture simplifies the implementation of network policies with a simple example. We will describe an actual implementation of a more complicated version of this scenario using FlowFlex in Section 6, as well.

Consider an enterprise network with three users: Alice, Bob, and Carol. Suppose that the network operator wants to implement the following policy: (i) HTTP traffic is limited to X Mbps for Alice (ii) VoIP calls for Bob should be forwarded without delay (iii) VoIP calls for Carol must be recorded. We consider how this policy might be implemented in an enterprise network, first using existing state-of-the-art technologies, and subsequently using FlowFlex.

Implementation with existing technology Figure 3 shows how a network operator might implement these policies today with the current state-of-the-art technologies and devices; effectively, the approach requires extensive deployment of expensive, special-purpose middleboxes, and redirection of all traffic through those middleboxes. First, the operator might deploy a packet header analyzer and content inspection device at the network boundary and pass all traffic through that device. Then, depending on the traffic type and policies matched, traffic is segregated into HTTP flows for Alice, VoIP calls for Bob, and VoIP calls for Carol. Alice's HTTP flows would then be redirected through a special-purpose traffic shaper. Carol's VoIP calls could be redirected through a specialized recorder device, whereas Bob's VoIP call are forwarded as is.

This approach has three shortcomings. First, all of the traffic must be re-routed through a content analyzer; this requires either extensive deployment of content analyzers, which could be expensive, or extensive rerouting through a smaller number of content analyzers, which could incur significant performance penalties. Second, changing global policies requires installing new policies in *every* content analyzer in the network, and ensuring that the policies on those middleboxes are coordinated properly with the associated traffic shaping devices. Third, implementing this relatively simple policy requires specialized middleboxes for each function, which increases management complexity, space and power requirements.

Implementation with FlowFlex FlowFlex takes a different approach, by effectively realizing a *centralized policy decision maker and distributed flow/packet level enforcer*. A centralized controller keeps track of policies and the corresponding logic blocks to enforce those policies. These logic blocks can operate at either the flow level or the packet level. The distributed switches that act as policy enforcers are stripped of most functionality except to match traffic to a class and execute the corresponding logic block. If the switch does not have the corresponding classification rule or logic block installed, the switch contacts the central controller to retrieve the appropriate logic block to forward traffic corresponding to that flow.

In our example case, the following sequence of events occurs. Figure 4 also illustrates these steps, where they occur, and how messages are passed between the switch and the controller.

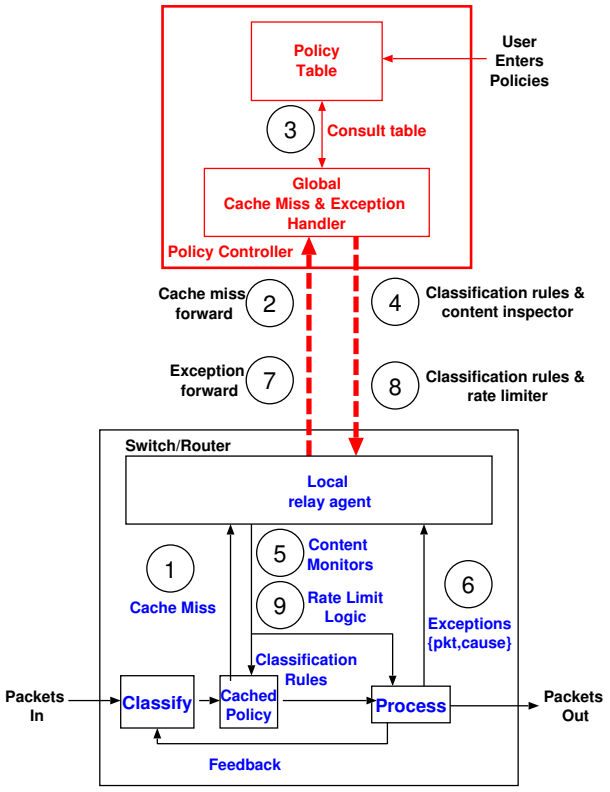

Figure 4: FlowFlex operation.

1. A network operator enters policies at the controller. Some time after this configuration, three new connections arrive at the switch: (a) an HTTP connection for Alice (b) a VoIP connection for Bob and (c) a VoIP connection for Alice. The classification phase tries to classify these flows to already existing rules. Since these are new connections, no classification rules from the policy table would match for the flows. Classification would then generate a cache miss and pass the packet to the local relay agent.
2. The traffic does not match any existing classification rule, so the local relay agent encapsulates the packet and sends it to the controller over a secure channel.
3. The controller has a global cache and exception handler with the view of network-wide events and data, as well as the policy table. By consulting these two databases, the controller determines with the best course of actions for the given flow.
4. The controller generates (a) the classification rules and (b) the corresponding logic block for the flow. It passes them to the relevant switch's local agent along with the packet.

   In this case, the controller would install the classification rule with that packet's headers (input port, source-ip, destination-ip, source-port, destination-port) and logic block to inspect the contents of the packets from that flow for the HTTP or SIP handshake. The controller would form the logic block so that it informs the controller in case the protocol is detected by generating an exception.
5. The local relay agent pushes these rules and logic blocks in the *cached policy* table and delivers the packet to the processing phase to apply these logic blocks. The processing phase would inspect that packet and any subsequent packets from that flow for the HTTP or SIP handshake.
6. Upon finding HTTP/SIP handshake, classify raises an exception, "*(packet, exception cause - HTTP/SIP protocol detected)*".
7. The local relay agent captures this exception and forwards it to the controller.
8. The controller will determine to which user this flow belongs. It will then decide the logic block to install by referring to the policy table and user quota. In this case, the controller will send back following three logic blocks: (i) A rate limiter to limit traffic to X Mbps for Alice (ii) A forwarding block without any delay for Bob's VoIP call and (iii) A replicating block for Carol's VoIP call that duplicates the packets and sends the copy to the controller.

   The controller sends back rules in the form *(classification rule for flows, processing logic block)*, together with the packet that triggered the exception.
9. The switch installs the processing block in the *cached policy* table and executed on that packet and any subsequent packets that belong to that flow. Packets are forwarded after the processing.

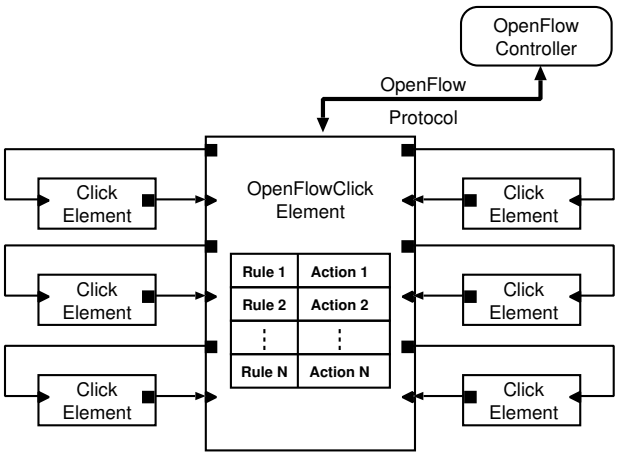

Figure 5: Element chaining using OpenFlowClick to create complex processing logic.

5. Implementation

The cornerstone of the FlowFlex design is an implementation of an OpenFlow [20] switch element for the Click software router [15]; this element is called *OpenFlowClick* [21]. This element allows us to seamlessly in-

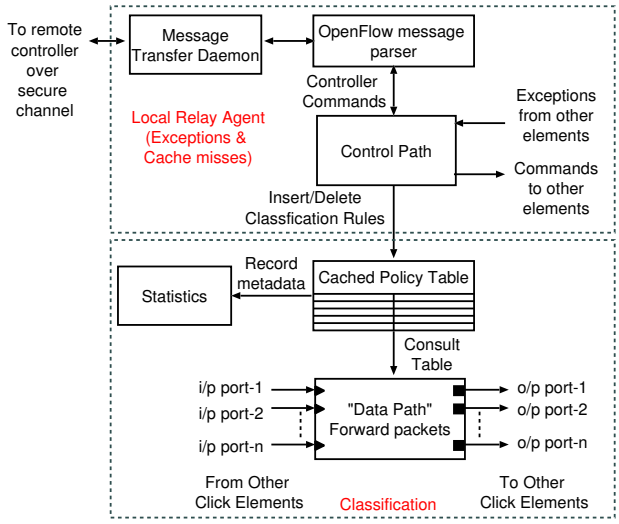

Figure 6: Architecture of the OpenFlowClick element.

tegrate the fine-grained packet processing offered by Click with the central, coordinated control provided by OpenFlow and the NOX controller [13]. We implemented the controller applications in Python on top of the NOX controller [13] framework.

Combining Click with OpenFlow allows users to dynamically create packet-processing function in the form of element pipelines. The rules installed in the OpenFlowClick element dictate which element chains a packet can traverse; a central controller can install these rules these rules.

We chose Click as the programmable software switch because prototyping using Click elements is fast. Click has a large library of existing elements that can be interconnected using an intuitive graphical language. The base library contains more than 200 elements that have implemented functions for IPsec, IPv6, wireless protocols, and many other functions. The tool is also easily extensible and mature, having been under development for more than ten years. Finally, Click provides good forwarding performance for a software router. A recent architecture based on Click with forwarding performance on the order of gigabits per second are already emerging [8]. Of course, the use of Click is limited by software forwarding rates, ultimately preventing FlowFlex from achieving the fourth goal of fast forwarding. However, recent work has shown that even software routers can be quite fast [8], and some developments are afoot to make Click port directly to a NetFPGA [16, 23], which could speed up the Click-based packet processing on the switches themselves.

The element chain Each of these elements connected to the OpenFlowClick element can be thought of as stages in the processing pipeline. We call this processing pipeline the *element chain*. By installing rules, the controller can increase or decrease the length of these element chains for each flow. Adding new functions for processing is as easy as installing a rule in the table to make a packet go through a particular Click element. For example, suppose we have an element chain with two Click elements: A NAT element which rewrites the IP addresses and keeps mapping between ports and internal IP addresses and a GRE element which adds a GRE header for tunneling. If an operator decides to add a policy that mandates that all the FTP traffic leaving the company must be secure, then all that the controller has to do is insert a rule that makes all FTP packets go through an encryption element after leaving the NAT element but before going through the GRE element.

Figure 5 shows some example element chains. Multiple Click elements can be connected to the OpenFlowClick's input and output ports, forming loops. Rules, installed by the controller, decide the output port for the packet. By deciding these output ports, these rules also decide the next Click element that will process the packet. Because the output of that element is fed back once again to the OpenFlowClick element, the controller can make the same packet pass through various elements by carefully choosing sequence of rules that will match with the packet after each processing step.

Figure 6 shows the architecture of the OpenFlowClick element. The main components are as follows:

- *Data path:* Matches packet headers against installed rules in the "Cached-Policy-Table" and forwards packets to other elements. This module constitutes the forwarding plane that matches packet headers and pushes them to other elements. Since this is a Click element, its input and output ports are in reality connections to other Click elements. Thus, there is no correlation between actual hardware interfaces of the switch and the number of ports belonging to the OpenFlowClick element. This arrangement allows to put other elements before and after the OpenFlowClick element.

- *Control path:* executes controller commands on the switch. Mainly, these commands install the rules in the "Cached-Policy-Table" and also invoke other elements' handlers. Element handlers make it possible to change the behavior of the Click elements without stopping the running router. An example could be: To increase a Queue's storage capacity, the controller might call the "capacity" handler with an argument equal to desired new capacity. The control path also accepts exceptions generated by other elements. These exceptions are transferred to the controller and thus give execution control back to the controller. The controller can then analyze the cause of the exception and decide how to handle it.

- *Statistics:* It records the meta data such as how many packets entered, how many matched the rules, how

many rules are present in the table, etc. This data is periodically reported to the controller.

- *Cached Policy Table:* It maintains the rules to match against packet headers. OpenFlowClick has two kinds of tables to store these rules: a linear table and a hash table. As the names suggest, searching a matching rule in the linear table is $O(n)$ operation whereas in the hash table, the matching time is $O(1)$. However, one cannot store rules with "don't care" fields in the hash table, but the linear table can store these rules. In practice, we think that the hash table would be used most of the time, since the controller forms the rules using all the packet header fields. Using the hash table significantly reduces the search time for the rule.
- *OpenFlow message parser:* This module takes care of parsing OpenFlow messages from the controller and calling functions with correct arguments from the control path to execute those commands. It will also encapsulate exceptions and control path responses into OpenFlow protocol messages and send them to the controller.
- *Message Transfer Daemon:* This module has two parts: one inside the kernel that takes data from the control path, and a second user space process that talks to the controller over TCP/IP sockets. The kernel part exchanges information with the daemon in the user space using the netlink interface.

We have also added functions to the basic OpenFlow protocol that are specific to the Click software router using vendor-specific extensions [2]. These extensions allow the controller to request the currently running Click configuration. Thus, a controller can know what elements are connected to each other by parsing the Click declarative language. It also allows the controller to hot-swap one Click configuration with another. We have also added a "invoke-element-handler" extension to the OpenFlow protocol, which, accepts the arguments "element-name", "handler-name" and "handler-arguments". This allows the controller to change behavior of elements connected in the Click configuration.

To facilitate exception handing, we have also provided a handler in the OpenFlowClick element so that other elements in the configuration can invoke it to generate exceptions. The general form of these exceptions is: (exception-name, cause, the packet causing the exception). These exceptions will then be forwarded by the OpenFlowClick element over vendor-specific extensions.

The next three sections describe three specific applications that take advantage of different parts of the FlowFlex processing pipeline.

6. Enhanced Classification

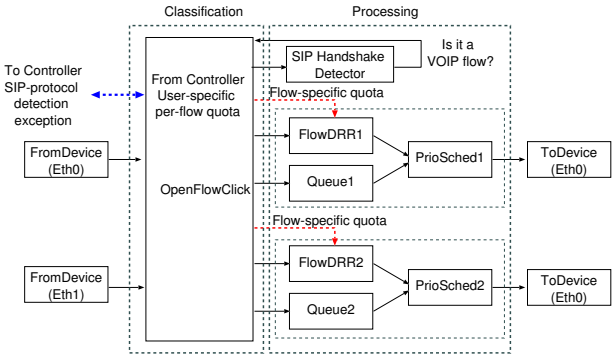

Figure 7: Enhanced Classification example. Click element configuration with OpenFlowClick element to talk to the controller. SIP detector classifies the flow as VoIP or ordinary flow. FlowDRR is deficit round robin scheduler for flows to enforce user specific quotas

In this section, we explore the additional capabilities that the *enhanced classification* features of FlowFlex provides. In particular, we revisit a more complex version of the problem that we posed in the introduction, where a network administrator wants to inspect packets for information about users associated with traffic flows, and then assign those flows to specific priority classes according to some centralized policy. Other protocols that could benefit from enhanced classification include port-agnostic protocol detection by analyzing packet data, and new protocols that reuse existing packet headers in new ways (*e.g.*, path splicing [19], routing deflections [26]).

FlowFlex enhances traffic classification by integrating following properties into a single, coherent framework:

- **Deep packet inspection (DPI):** Packet content inspection along with headers.
- **Distributed Coordination:** Correlation with other network-wide data such as other users currently logged into the nodes (to associate flows with users), current SYN rate and whether it exceeds some threshold (to categorize it as potential DDOS attack).
- **Cache Propagation:** Saving classification effort at other switches further down the path. Once the entry point switch classifies the traffic by inspecting the contents, the controller can automatically insert rules at other switches even before the traffic arrives.
- **Globally consistent classification:** An administrator can change the policy and force switches to reclassify the traffic with the new policy, at any time.

These properties make the classification phase policy-aware and powerful. In contrast, current techniques mainly rely on packet headers for classification and even if they inspect the contents, they cannot correlate with information available on other nodes due to lack of a central coordinator.

6.1 Motivating Application

To illustrate the classification feature of FlowFlex, we present an example motivated by our own campus network admins: specialized per-user VoIP traffic management. Such a system requires that each user is assigned a certain VoIP call quota for a given time interval. Additionally, certain groups of users, such as faculty members, might be given preferential treatment over others like students. Thus, if students' VoIP calls are in progress and a faculty member starts a call, then packets from a student's flow are dynamically re-assigned to a lower priority queue. Detecting the user to whom the flow belongs, requires associating users logged in to the nodes, and flows belonging to nodes.

Exiting approaches cannot associate flows with the users themselves and efficiently relate them to some centralized policy. At best, we can manage such an association by associating DHCP server logs with the IP addresses in the flow. However, such an association per flow at each device is cumbersome to manage. Also, it exposes one more dimension user-name at each switch, adding extra programming complexity. A second point is that the traditional system's classification rules are somewhat rigid. Once a flow is classified into a category, it will most likely remain in that category for its lifetime.

On the other hand, FlowFlex's ability to perform a combination of fine-grained processing at switches with centralized policy at the controller makes this application easy to develop. The FlowFlex switches identify the application-level protocol in the flows and can also associate them to the user with the help of a user-to-nodes association table at the controller. Maintaining such table and keeping track of users logging in and out has been discussed in Ethane [5]. This mode of operation reduces complexity at the switches as they have to execute rate-limiting functionality for a particular flow, indicated by the controller, without bothering about doing flow-to-user coordination.

We now describe how FlowFlex sets the rules to make automatic VoIP classification with minimum controller involvement. First, the controller exports a processing logic block at setup time to inspect the packet contents for the unclassified packets, and to raise an exception if the contents are found out to be VoIP. Second, the switch's classification phase tries to classify new incoming flows with pre-existing classification rules using packet headers. If the switch fails to classify the packets into any pre-existing class, then it will classify the packets into unknown-class and pass on to the processing phase. The processing phase maintains a small amount of state to record any changes due to each new packet in either direction until the class is decided to be VoIP or something else. If the processing phase determines the class to be VoIP, then it raises an exception and sends the packet to the controller along with *voip-class-found* exception.

When the controller receives the exception, it associates the username with an associated quota and determines whether the user has exceeded the quota. If it has not, then the controller creates a new classification rule for that flow, as well as the corresponding processing logic that resides in the switch. In this case, the new logic block forwards the packets, keeps track of bandwidth usage for that user, and reports it periodically to the controller. If a faculty member flow is detected, then after setting the classification rule and logic block for that flow, all the logic blocks for the students flows on the switch are invalidated. Subsequent packets in students' flows will cause cache misses and will be sent to the controller. At this point, the controller will re-assess the policy and that student's remaining quota to create a correct logic block.

6.2 Implementation

We have implemented a NOX controller application, "voip-campus", that keeps track of user login information and bandwidth consumed by the user in that time interval. We use Click in conjunction with the OpenFlowClick element to implement the switch functions. To associate traffic flows with specific users, we have also written a custom classifying element that detects the SIP handshake and looks into the traffic to determine the associated user.

Figure 7 shows the Click configuration at the switch for this application. FlowDRR is another custom element that we have implemented to schedule and rate-limit the traffic belonging to a particular user. This element exports handlers that allow the controller to change deficit quantum associated with any flow and thus tweak the user specific rate. Packets first enter the OpenFlow Click element that matches packet headers with pre-existing classification rules. If the packet does not match any of the classification rules, then it is redirected to the SIP handshake detector. If the handshake is not found, then the packet is forwarded as a normal layer-two switch would forward it. If the SIP handshake is detected, then the OpenFlow Click element is notified that in turn generates an exception SIP-handshake-detected and notifies the controller. The controller application then installs the classification rules in the OpenFlow Click element to redirect it to FlowDRR1 or FlowDRR2. It also sends the deficit quantum information for that flow by checking who that flow belongs to. Non-VoIP traffic is deposited to either Queue1 or Queue2 depending on its output port. Two PriSched elements schedule traffic on each of the output port. They give higher priority to the VoIP traffic.

FlowFlex allows network operators to implement this solution quickly and in only a few lines of code. We wrote SIP-detector from scratch, but we modified the DRR packet scheduler already present in the Click elements library. A Controller side Python application maintains the user-login and per-user-quota information. Currently, all the elements that one might possibly want traffic to pass

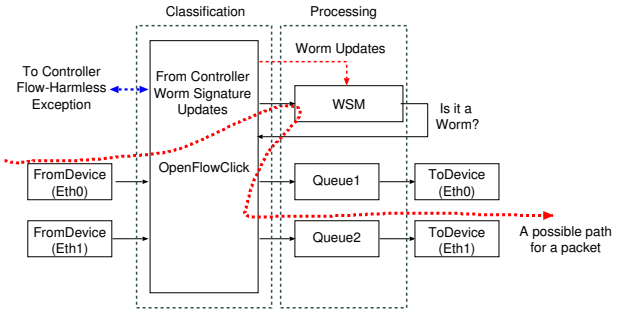

Figure 8: Policy exception example. WSM detects if there is a malware. If that the malware is found then the flow gets dropped. If there is no malware then flow will no longer need to go through WSM. In either case, exception is raised to decide fate of the flow

through must be specified in the Click configuration file and have to be connected to the OpenFlowClick element. Thus, if one decides to start encrypting VoIP calls for a certain group of users (*e.g.*, administration), then an encryption element has to be present in the Click configuration. With the help of dynamic element loading (which we discuss at more length in Section 9), it is possible to load such an element at runtime.

7. Policy Exceptions

As described in Section 3, FlowFlex can also generate distinct exceptions at different switches but process them centrally at the controller. One application scenario where this function may be particularly useful is for distributed intrusion detection systems (IDS). Typically, a network administrator will want to specify a coordinated, centralized policy for detecting offending traffic but implement that policy across a set of distributed switches. This *distributed IDS* problem is challenging today, mostly because existing approaches provide no clean way to coordinate distributed, packet-level inspection with a centralized policy without deploying of expensive, custom, domain-specific hardware. In contrast, FlowFlex's *policy exceptions* make distributed IDS much easier to implement by providing an explicit control path between distributed switches and a centralized controller.

Policy exceptions are controller-defined events that, upon triggering at the switch, pass the processing control for that flow to the controller. One can think of an analogy to debugging using watch-points with a program such as the GNU debugger (gdb) [11]: A user may set a watch-point such as *when 'i' becomes 5* and let the processor continue execution without user's involvement until the watch-point becomes true. Thereafter, the user can single step through the code. Similarly, a FlowFlex-enabled switch can raise an exception set by the centralized controller when network traffic generates such an event.

This processing mode allows FlowFlex to provide the following important properties:

- **Centralized policy management with peripheral enforcement:** Policies are managed centrally, and switches take care of enforcing them on per-packet. Thus controllers can program all classification rules and processing logic with exception triggering conditions at the switches. After that switches take care of handling and processing traffic. The controller only does the job of maintaining a consistent view across the network.
- **Delegation of traffic processing at a switch:** Traffic is processed at the peripheral switches most of the time.
- **Scalability:** Since most traffic handling is done at the switches, the controller is less loaded and handles only exceptions.

The combination of these properties enables easy deployment of new custom applications, such as the distributed IDS application that we describe below. Policy exceptions can also be applied other applications, such as implementing packet sampling only when an exception is raised after seeing a particular number of packets, or after matching a particular type of packet content. This could also be used to diagnose network anomalies such as loops. If a router begins to drop many packets because of TTL reaching zero, then it could also raise an exception and send the packet to the controller, at which point the controller could take some corrective action (*e.g.*, it might alert switches about the presence of a loop, or even take a proactive style approach to correct the loop, as in RCP [4, 10] or 4D [12]).

7.1 Motivating Application

Enterprise networks are constantly exposed to a variety of attack traffic. These attacks range from connectivity disruption of the Enterprise network to the outside network via DDOS attacks to worms that either bring the internal network to its knees by consuming resources or spyware that steals the company's confidential intellectual property. However, deploying these solutions using existing technologies has several shortcomings. First, detection may require specialized middleboxes, such as application firewalls or intrusion detection systems; this extra machinery introduces additional complexity and management overhead, and also consumes precious power and rack space for an overly specific function. Second, even if the network administrator were to deploy these boxes, they cannot inherently detect a coordinated attack on different parts of the infrastructure that an architecture like FlowFlex, which has a coordination element, could detect.

An alternate design—which we adopt with FlowFlex—would instead redirect suspicious traffic to a smaller set of specialized devices for further inspection, sending "normal" traffic directly reach the host. In such a system, any new flows would be subjected to a lightweight inspection near ingress points which would focus on efficiency and fast processing, but take care to keep false positives

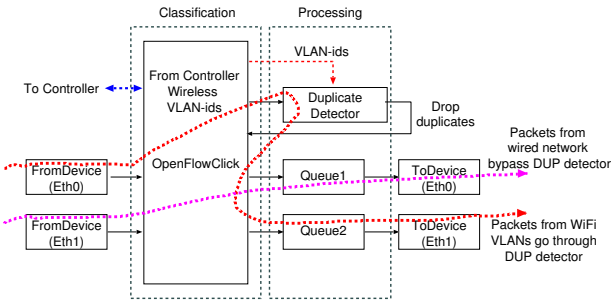

Figure 9: Custom processing example. Duplicate-detector is a custom Click element that detect duplicate packets using bloom filter and drops them. Only packets belonging to WiFi VLANs need to be subjected to such processing

low. Traffic that passes this scrutiny can be sent directly to the respective destination hosts. Further, if this inspection concludes that some subset of traffic should be excluded from further inspection because it is deemed to be "safe" might indicate this decision to the ingress points to save unnecessary additional processing. If inspection deems the traffic to be suspicious, then the traffic might be subjected to additional actions (*e.g.*, more intensive scrutiny, filtering).

7.2 Implementation

We have implemented this using a worm-detection application in the NOX controller and a worm-signature matcher element, `WSM`, in Click. The Click configuration for the switch is shown in Figure 8. Any new incoming flow are sent to the controller. The controller extracts the headers and makes a classification rule to send the traffic through the WSM element.

The WSM element attempts to match the packet contents against the database of worm signatures; this database could be periodically updated from the centralized controller. If WSM detects that the flow does not contain any suspicious content after a certain amount of traffic, then it raises an exception to the controller. The worm detector application at the controller will then declassify the flow and change its processing logic to simply forward rather than inspect and forward. On the other hand, if WSM matches any worm signature, then the flow is redirected for further analysis or quarantine or is simply dropped. This application could be further extended to detect DDOS attacks where a Click element detects if its getting SYN packets with a rate higher than threshold.

Our code for the WSM element is about 600 lines of C++ (for signature check & signature database) and corresponding Python application at the controller is about 400 lines long. Both were developed and tested under 30 hours.

8. Custom Processing

We can think of the element chain as a dynamic pipeline with multiple stages; the controller can inject or remove a logic block at any stage in the processing pipeline. The controller determines which logic blocks should be installed at each switch, as well as the order in which they would execute on the network traffic. This custom processing allows for central control of the flows, dynamically and according to a global policy, but facilitates custom processing of network traffic by applying arbitrary logic blocks *at the switch*.

FlowFlex's custom processing provides the following salient features:

- *Flexible, customizable switches:* Switches in FlowFlex can execute many logic blocks. These simple logic blocks permit more complex functionality.

- *The ability to dynamically change the processing flow for packets:* It is possible to inject functionality dynamically by just adding additional processing stages the pipeline. For example, if a network flow is being processed with two blocks (*e.g.*, malware detection and tunneling), then other features could also be added (*e.g.*, inserting an encryption block in the processing pipeline).

Customized processing is also present in other architectures such as network processors, NetFPGA, and Click. None of these distribute processing solutions inherently allow a centralized controller to upload logic blocks in the pipeline. We believe that giving the centralized controller complete power over the execution paths, allow to make co-ordinated decisions. In platforms like NetFPGA, there is also a limit to how much customized processing blocks can be uploaded on a single board.

The ability to integrate custom packet processing based on global policy may be useful for many applications. Applications that might find this processing helpful include: selective tunneling or encryption of flows, conversion of protocols in the network, and forwarding according to link utilization in the network. Other groups have already started using FlowFlex's customized processing capability of FlowFlex, by combining Click's encapsulation element with the OpenFlowClick element [27].

8.1 Motivating Application

Let's focus on the problem of eliminating duplicate packets in the network. In this case, FlowFlex can be used to eliminate redundant or duplicate packets, as previous work has motivated [3]. Any packet going through lossy links can be duplicated at the egress points automatically. Wireless networks such as mesh networks can benefit from frame duplication to add redundancy and mitigate the effects of link level losses. However, when such packets reach the wired network, duplicate packets must be eliminated at the ingress switch to avoid consuming

excessive bandwidth. Currently, most of the time, packet recovery is the job of a higher-layer protocols such as TCP.

One way to manage these frame losses is to duplicate the frames at the ingress switch and send them along two different routes in the wireless network. However, when these packets reach a common gateway in the wired network, they should eliminate the duplicate packets.

8.2 Implementation

FlowFlex subjects traffic coming from wireless VLAN to a duplicate detection logic block. The mechanism used in the logic block could be any of the standard hash based detection mechanisms mentioned in recent work on redundancy elimination [3, 24]. We have implemented this function by implementing a Bloom filter in Click. This element is then connected to the OpenFlowClick element, as shown in Figure 9. We wrote a small NOX controller application, "dup-detect", that installs a rule in the OpenFlow Click element to send any packet with VLAN-id equal to that of wireless VLAN to the bloom filter element. If the packet is a duplicate, then it is dropped. Otherwise, it is fed back to the OpenFlowClick element for further processing.

9. Future Work

FlowFlex opens several avenues for future work; we discuss two possible extensions to the existing FlowFlex framework. The first extension is the ability to dynamically load and unload Click elements on the FlowFlex switch itself, to enable additional flexibility. The second extension is to combine the FlowFlex architecture with more customizable hardware-based processing on the network devices themselves. In this section, we discuss each extension in turn.

Dynamic loading of packet processing elements The OpenFlowClick can send packets out different output ports. This allows the controller to install rules to choose between various processing element paths. However, those elements selected to execute on the packets have to be loaded before the Click router is instantiated and starts running. That is, those elements have to be mentioned in the Click router configuration file and be connected to the OpenFlowClick element directly or via some other elements. Click has more than 200 elements, so one cannot connect all these elements to the OpenFlowClick element in the Click router configuration. This is because, it might not be known what elements are required before FlowFlex starts running or it might not be practical to connect them all at once just in the case that an element might be required at some point in the future. One option to solve this is to stop the router and start it with a new configuration that has the this required element. Another, slightly better alternative is to hot-swap a new configuration. Hot-swapping could cause loss of packets that are currently getting processed in the Click router. A network device, with high traffic load, cannot afford such losses.

A solution for this is to add ports dynamically to the OpenFlowClick element and loading a newly required element between these ports. One can imagine a situation where exception thrown by the switch causes the controller to form logic blocks that are not supported by the Click's current configuration. In that case, controller can first request for the Click configuration. After detecting that the required element is not present in the currently loaded configuration, controller can send a command to first add new port pair to the OpenFlowClick element and then to load the required element between this pair.

One can think equivalence between this solution and loadable kernel modules to design highly available system. Such functionality would require the Click software router to support dynamic loading and unloading of elements. Although our current implementation does not support this feature, we are working on enabling this function in our system.

Integration with programmable hardware pipelines The current FlowFlex architecture still involves integration of an OpenFlowClick element with Click processing elements. The current instantiation of FlowFlex thus requires the FlowFlex switches themselves to be instantiated in software, thus preventing some of the benefits of distributed processing from being realized (namely, the ability to process packets in hardware).

In future work, we aim to accelerate the processing on switches by implementing common data-plane functions in programmable hardware (e.g., NetFPGA [17]), which in turn can raise software exceptions to Click or interface directly with the OpenFlow flow-table rules. We envision a two-level processing hierarchy, whereby the most common functions are implemented in programmable hardware, primary exceptions are handled by Click in software, but locally on the switch, and secondary exceptions are handled by the controller.

10. Conclusion

This paper has presented FlowFlex, a new programming model for network systems and protocols. Today, operators and protocol designers must make difficult decisions between the coordination offered by centralized control and the performance that can be achieved by implementing policies across devices distributed across the network. As a result, they also make trade offs between the flexibility and deployment speed of software-based solutions and the speed of hardware. The design of FlowFlex aims to offer developers the ability to realize centralized, coordinated policy as distributed implementations across the network. The model marries the convenience of flow-based processing with the flexibility of packet-based processing; software exceptions raised at switches and logic blocks

passed from central controllers to the switches also combine the benefits of centralized control with distributed implementation and realization.

To demonstrate the utility, flexibility, and ease of development that FlowFlex provides, we have implemented three network applications using FlowFlex and shown how the resulting implementation is more flexible than that which could be accomplished with existing state-of-the-art programming paradigms.

Stepping back, we hope that FlowFlex places software-defined networking in a new light. Previously, protocol designers did not have a way to quickly develop protocols with coordinated, fine-grained control and reap the performance benefits of hardware implementations. We believe that successful FlowFlex prototypes will translate readily into deployable systems: individual forwarding elements model the hardware pipeline structure common to ASIC designs and the global logic in controllers is trivially ported into a production data center. Thus, we see this general model—fast, inline packet forwarding with flexible, on-switch logic blocks and off-switch software exceptions—as perhaps a generic model for designing next-generation switches that are designed with innovation as a top priority.

REFERENCES

[1] NOX: An OpenFlow controller. `http://www.noxrepo.org`.

[2] OpenFlow Specification v0.8.2. `http://yuba.stanford.edu/openflow/documents/openflow-spec-v0.8.2.pdf`.

[3] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: The implications of universal redundant traffic elimination. In *Proc. ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[4] M. Caesar, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proc. 2nd USENIX NSDI*, Boston, MA, May 2005.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane : Taking control of the enterprise. In *SIGCOMM '07*, 2007.

[6] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[7] Cisco IOS Master Commands List, Release 12.3. `http://cisco.com/univercd/cc/td/doc/product/software/ios123/123mindx/crgindx.htm`.

[8] M. Dobrescu, , N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[9] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proc. 2nd Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005.

[10] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and K. van der Merwe. The case for separating routing from routers. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture*, Portland, OR, Sept. 2004.

[11] GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb/`.

[12] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *ACM Computer Communications Review*, 35(5):41–54, 2005.

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, July 2008.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[16] C. Kulkarni, G. Brebner, and G. Schelle. Mapping a Domain Specific Language to a Platform FPGA. In *Proceedings of the 41st annual conference on Design automation*, pages 924–927, 2004.

[17] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *IEEE International Conference on Microelectronic Systems Education*, pages 160–161. IEEE Computer Society Washington, DC, USA, 2007.

[18] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM*, pages 3–17, Pittsburgh, PA, Aug. 2002.

[19] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path Splicing. In *Proc. ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[20] OpenFlow Switch Consortium. `http://www.openflowswitch.org/`, 2008.

[21] OpenFlow Click element. `http://www.openflowswitch.org/wk/index.php/OpenFlowClick`, 2009.

[22] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. Internet Engineering Task Force, Jan. 2006. RFC 4271.

[23] G. Schelle and D. Grunwald. Cusp: A Modular Framework for High Speed Network Applications on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 246–257, 2005.

[24] N. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. ACM SIGCOMM*, Stockholm, Sweden, Sept. 2000.

[25] J. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, et al. Supercharging PlanetLab: A High Performance, Multi-application, Overlay Network Platform. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.

[26] X. Yang, D. Wetherall, and T. Anderson. Source selectable path diversity via routing deflections. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.

[27] M. Yu and J. Rexford. Hash, don't cache: Fast packet forwarding for enterprise edge routers. In *Proc. Workshop: Research on Enterprise Networking*, Barcelona, Spain, Aug. 2009.